ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4 July 1997

Paper:

Tactics for Transformational Programming

Rosemary Monahan and Franz Geiselbrechtinger

Published in collaboration with the British Computer Society

BCS

Springer

Tactics for Transformational Programming

Rosemary Monahan
Department of Computer Science, University College Dublin
Belfield, Dublin 4, Ireland.

Franz Geiselbrechtinger
Department of Computer Science, University College Dublin
Belfield, Dublin 4, Ireland.

Abstract

This paper discusses the relationship between transformational programming and theorem proving. It illustrates the use of the theorem proving environment as a basis for a program construction tool DEBATE [1] (Deduction Based Transformational Environment) which is under construction in University College Dublin.

Using a theorem proving framework directly would require the user to be familiar with theorem proving details. The tool user should only be concerned with transformational programming steps and not with theorem proving activities. Therefore a layer of transformational tactics are discussed and presented. These tactics consist of the application of theorem proving tactics. However, they ensure that the user's only interaction with DEBATE are design decisions required within the transformational programming paradigm. The N Queens problem is used throughout the paper to demonstrate how the Isabelle theorem prover is adapted by a transformation tactic layer so that it may be used as a program construction tool.

1 Introduction

Transformational programming is a method for constructing programs from their specifications. A series of rewrite steps are applied to a problem specification with the aim of constructing its operational solution. Each rewrite rule preserves the correctness of the program development with respect to the initial specification. It transforms the specification into one which is more algorithmic and hence closer to an operational solution.

This particular view of the software development process was the main focus of the CIP project [1] and is presented in detail in Partsch [2]. Other similiar work is discussed in Feather [3] and in ProSpecTra [4].
Advantages of this approach to program development include

- Correctness by construction
- Capture of programming knowledge in explicit reusable rules
- Provision of documentation of design decisions

Another major advantage of this method is that it lends itself to tool support. This is due to the formal nature of the problem specifications and the rewrite rules. The development of such tools has been the product of projects such as CIP, ProSpecTra, KORSO [5] and KIDS ([6], [7], [8]).

In DEBATE we exploit the similarities between program construction and theorem proving to construct a tool for program development by the method of transformational programming. This tool provides support for the concrete, efficient implementation of specifications. A first attempt at building such a tool in University College Dublin was a Prolog based system [11]. This system was not supported by a theorem prover. In DEBATE we use the theorem prover Isabelle to support and to automate the process of transformational programming.

[1] Work was in part supported by FORBAIRT grant SC/95/408

In this paper we commence with a brief discussion on the method of transformational programming. The correspondence between the steps in this process and those in theorem proving are discussed. We consider problems which arise as a result of using the theorem proving environment directly as a program construction environment. Finally, we illustrate how these problems may be overcome through the development of tactics for transformational programming.

2 Transformational Programming and Theorem Proving

In transformational programming a program is constructed from its specification through the application of a series of rewrite rules. The goal of this rewriting is the production of an operational solution to the problem. An individual step in the transformation process consists of focusing on part of the specification, choosing an appropriate rewrite rule and applying that rule.

In general the rewrite rules are of the form

If C then A $\longrightarrow$ B

where

- A is the syntactic template for the source to be transformed (abstract)
- B is a syntactic template for the target to be obtained (concrete)
- C is the semantic template for the applicability conditions of the rule

Such a rule indicates that an instance of A can be transformed into an instance of B if the condition C is true. The applicability conditions consist of the semantic and syntactic constraints on the rule application. They ensure that the specification obtained as a result of the rule application is consistent with the previous specification. Hence, the final program obtained will be a correct implementation of the original specification. When a rule is applied another part of the specification is focused on and the process is repeated. This continues until an implementation of the problem has been obtained.

In a program construction tool which supports transformational programming, a theorem prover may be used to evaluate the applicability conditions. It may also be used to automate the development process. To achieve this the problem specification, the rewrite rules and methods of applying these rules must be available in the theorem proving environment. The theorem prover Isabelle supplies resolution for application of the rules. Rewrite rules may be formalised and proved using Isabelle tactics and the problem specification may be written in the language ML. ML is the language used by Isabelle.

The specifications we are concerned with are algebraic specification which consist of a signature and axioms. The signature describes the operators, constants and types which are premitted within the specification and the axioms define the behaviour of these operators. When a transformation begins an axiom from the problem specification is chosen for development and transformed by the application of the rewrite rules. Within DEBATE the rewrite rules are applied as Isabelle inference rules. This will be discussed further in section three but we first distinguish between two types of rules which may be applied to specifications - synthesis rules and simple rewrite rules.

2.1 Synthesis Rules

Synthesis rules capture problem solving techniques. They are applied throughout the development process to transform specifications into ones closer to an operational solution. An example of a synthesis rule is the case introduction rule

If Conditions then
Problem $\longrightarrow$ if Guard then Subproblem 1 else Subproblem 2

Using this synthesis rule transforms the original problem into two subproblems if the rules conditions are true. As there are additional assumptions supplied by the guard, it may be possible to produce a solution to these subproblems by simple rewriting. Otherwise, another synthesis rule is applied.

Characteristics of the synthesis rules are that

- New subproblems are generated
- User interaction may be required.

An example of user interaction in the case introduction rule is that the user must supply the guard. This interaction supplies a programming design decision in addition to the design decision of which rule to apply.

2.2 Simple Rewrite Rules

Simple rewrite rules consist of simplification rules such as

If True then A else B $\Longleftrightarrow$ A.

Their application causes the specification to be rewritten without the introduction of new subproblems. They include simplification rules which are obtained from the specification axioms. These are mainly rules which define the data structure on which our problem is based. The logical basis for these rules is substitution using equals.
The principle roles of simple rewrite rules are

- Modification of the specification to allow application of synthesis rules.
- Assisting the proof of applicability conditions
- Rewriting solutions for efficiency reasons

An example of the latter is the linear resursion to loop transformation rule in which no subgoals are generated but the solution is expressed in a more efficient manner.

2.3 Transformational Programming and Theorem Proving - A Comparison.

Program construction by transformation may be considered as the task of constructing a development tree. The root of the tree represents the original problem specification while the subtrees represent the subproblems which are generated by the application of synthesis rules. Leaf nodes correspond to subproblem solutions which when combined generate an operational solution to the overall problem. Simple rewrite rules may be applied to the nodes of the tree but result in their reorganisation as opposed to subproblem generation.

Theorem proving uses the principle of backward proofs by resolution. Proofs start with the theorem to be proved as the goal. This goal represents the initial proof state. It is equivalent to the initial specification to be transformed in the program construction tree. A goal is proved by a series of resolution steps which apply inference rules to the proof state. This is equivalent to the application of rewrite rules in program construction. Given a proof state with subgoals

G_1, ..., X, ... G_n

and an inference rule of the form

B_1, B_2, ..., $B_n \Longrightarrow$ A

where A and X are unifiable, then the inference rule is applicable to goal X. The proof state generated as a result of resolution is

G_1', ... , B_1', B_2', ... , B_n' , ... G_n'

where G_i', B_i' are subgoals obtained by substitution according to the unification of A and X.

In theorem proving a subgoal is satisfied by simplification to true. Otherwise another resolution step is applied. In transformational programming a specification is satisfied when all nodes representing subproblems are developed into operational solutions. These solutions are then combined to give the solution to the overall problem.

In summary, the correspondences between the program construction process and theorem proving activities are presented below. These correspondences are as a result of a more detailed discussion in [13].

Transformational Programming	Theorem Prover
Problem / Focus	Goal
Matching	Unification
Synthesis Rule application	Resolution
Simple Rewrite Rule application	Simplification
Subproblems and Conditions	Subgoals
Proving Applicability Conditions	Satisfy Subgoals
Focusing	Choosing Subgoal
Instantiation	Applying Unifier

From these similarities it seems plausible to use a theorem proving environment as the basis for a program construction tool. Using the Isabelle case studies have been undertaken to determine whether this approach is feasible. The problems studied include Warshalls algorithm [13] and the Prefix problem [14].

Isabelle ([15], [16], [17] and [18]) is a generic theorem prover which provides a framework for developing proof systems. It is written in the functional language ML and supports proof of theorems using the rules of natural deduction. Isabelle provides a large repository of tactics for resolution and simplification purposes.

The use of Isabelle tactics during program construction force the programmer to become familiar with theorem proving tactics and Isabelle commands. This low level of interaction with a program construction tool is both unnecessary and complicated. Isabelle has a vast amount of rules and tactics which make it a powerful tool but also make it difficult to use for program construction. This is because the user must determine which Isabelle tactics and commands achieve the required goal as well as concentrating on the task of program construction. Many of the activities required during program construction may be automated. We can simplify the concerns of the programmer by concealing the theorem proving activities. This is achieved by building a new level of program construction tactics on top of the existing Isabelle framework. These tactics take the interaction between the tool and the programmer to a level which is concerned with the tasks in program construction only. In the section which follows we discuss and present some of these tactics.

3 Transformational Programming Tactics.

In this section we consider some tasks in program construction and present transformation tactics to achieve them. Those presented are

- Setting up the Problem
- Design Steps
- Simplification
- Closing Developments

Extracts from the transformational solution to the N Queens problem [2] are used as examples throughout the discussion. Informally the N Queens problem can be stated as

Is it possible to place N Queens on a N*N chessboard in such a way that they do not attack each other ?

For two queens to not attack each other they must not be placed in the same row, the same column or the same diagonal. Formally this is specified as [2]

$$queens(n) = \exists\ s\ .\ ||s|| = n \wedge nconf(s)$$
$$nconf(s) = \forall\ i.\ \forall\ k.\ i \neq k \longrightarrow s[i] \neq s[k] \wedge |k-i| \neq |s[k] - s[i]|$$

[2] The notation ||s|| represents the length of s and |k-i| represents the absolute value of k-i

The predicate queens(n) checks if there is a sequence of n positions which fulfil nconf(s). The predicate nconf(s) determines a sequence of positions such that queens in these positions do not attack each other. This specification will be used in the following sections to demonstrate tactics for setting up and commencing the construction of an operational solution to a problem within the tool environment.

3.1 Setting up a Problem

The initial step in tool based program construction is to set up the problem. This involves

- Formalising the problem and relevant data structures in some specification language and
- Selecting the problem to commence development.

The specification tool ASPECT, constructed in University College Dublin ([9], [10]), assists the building of algebraic specifications in the language SPL [19]. It will be integrated with our development tool DEBATE so that implementations of these specifications can be developed automatically. To use Isabelle the problem description must be transformed into Isabelle specific constructs. This involves formalising the problems data structures in the language ML and setting up the problem as the initial proof state. Formalising a problems data structures is done automatically by a translation from SPL specifications to their ML representations. The task of setting up the problem as the initial proof state will be achieved by a set up tactic which is supplied by DEBATE.

When setting up a problem for development the user's only concern should be to supply the problem definition and its associated theory. The theory of a problem describes the context within which the proof is to be developed i.e its specification and associated data structures.

A problem is specified as

```
I (x)⇒ O (x, f (x))
```

where I is the input predicate and O is the output predicate relating the input x to the output f(x). It is focused on in Isabelle by writing

```
goal Problem.thy "I (x)⇒ O (x, f (x))"
```

The problems associated theory is in the file Problem.thy which must be loaded before the goal is set up. This brings all the definitions in that file into the Isabelle environment in which the development is taking place.

Example:
In Isabelle the N Queens problem is formally specified as

```
NQueens.thy = Seq +

consts

    queens   :: " nat    ⇒ bool"
    nconf    :: " 'a Seq ⇒ bool"

rules

    queens   "queens(n) = ∃ s .  ||s|| =n ∧ nconf(s)"
    nconf   "nconf (s)  =  ∀ i.∀ k .i ≠ k ⟶ s[i] ≠ s[k] ∧ |k-i| ≠ |s[k] - s[i]|"

end
```

This file is loaded by typing

```
use_thy "NQueens"
```

at the Isabelle interface. This specification is based on the sequence data structure Seq . The problem is focused on by setting the part of the specification we want to modify as the goal. The definition of the operator queens is focused on for development as follows

goal NQueens.thy "True $\longrightarrow$ queens(n) = " $\exists$ s . ||s|| =n $\wedge$ nconf(s)";

where NQueens.thy is a file containing the problems theory as above.

As the user should not be concerned with Isabelle actions but with transformational activities, a transformation tactic is written to set up the problem as above. This tactic takes the name of the theory used, the name of the function to be developed and that functions definition as parameters.

An example of using this tactic is in setting up the NQueens problem where the user inputs the command

Setup(NQueens, queens, "True $\longrightarrow$ queens(n) = $\exists$ s . ||s|| =n $\wedge$ nconf(s)";

causing the file NQueens to be loaded and the definition of queens(n) to be the initial proof state. This proof state is as follows

1. "True $\longrightarrow$ queens(n) = " $\exists$ s . ||s|| =n $\wedge$ nconf(s)".

where subgoal 1 is the function definition to be transformed and True $\longrightarrow$ informs us that there are no assumptions which must be true for the definition to be valid. Use of this tactic ensures that the user's interaction with the development tool is strictly in line with the method of transformational programming.

3.2 Design Steps

A design step consists of the application of a problem solving technique to a specification. These problem solving techniques are captured by synthesis rules. We begin the discussion by explaining how a synthesis rule would be applied on the Isabelle level. We then consider modifications which can be made to develop a design tactic to conceal the theorem proving interface from the user. Firstly the representation of the synthesis rules as Isabelle inference rules is introduced.

Representation of Synthesis Rules:

The principle tactic in Isabelle is resolution. As developments are done using backward proofs by resolution, rules are interpreted in the opposite direction to which they are written. The synthesis rule

if C then A $\longrightarrow$ B

is represented as an Isabelle inference rule in the format [| C,B |] $\Longrightarrow$ A. This states that if the condition C evaluates to true the concrete representation B implies the abstract representation A. In program construction terms this means that the specification represented by A may be replaced by the specification represented by B.

Within the theorem proving environment a transformation theory is set up representing these rewrite rules as inference rules written in ML. They are proved using Isabelle tactics and are therefore available in Isabelle for use as safe program construction rules. This transformation theory must be executed before the program construction commences.

As discussed earlier characteristics of synthesis rules are that

- New subproblems are generated
- User interaction may be required.

Therefore, synthesis rules must be represented in a form where the solutions to the subproblems are not eliminated by proving as they would in a theorem prover. Subproblem solutions must be maintained to give a program which implements the initial specification. In addition, one of the subgoals generated from the application of a synthesis rule must be a proposition which expresses the overall solution as a composition of the subproblem solutions. Therefore,

the general format of a synthesis rule is

$$\text{If } C_1, \ldots, C_n \text{ then } I(x) \Rightarrow O(x, f(x)) \longrightarrow$$
$$I(x) \Rightarrow (I_1(x) \Rightarrow O_1(x, s_1(x)), \ldots, I_n(x) \Rightarrow O_n(x, s_n(x))) \wedge$$
$$f(x) = F[s_1(x), \ldots s_n(x), x]$$

where I is the input predicate and O is the output predicate relating the input x to the output f(x). The transformation of

$$I(x) \Rightarrow O(x, f(x))$$

into

$$I(x) \Rightarrow (I_1(x) \Rightarrow O_1(x, s_1(x)), \ldots, I_n(x) \Rightarrow O_n(x, s_n(x))) \wedge$$
$$f(x) = F[s_1(x), \ldots s_n(x), x]$$

may occur on the condition that $C_1, \ldots C_n$ are true. $O_i(x, s_i(x)) : 0 < i \leq N$ are the subproblems produced by the rule application. The predicates $I_i(x) : 0 < i \leq N$ are the preconditions to determining the solution to S_i. The function f(x) solves the problem and is expressed as a composition of the subproblem solutions S_i. The template for a synthesis rule is written in Isabelle as an inference rule of the form

$$[\!| C_1, \ldots, C_n \wedge$$
$$I(x) \Rightarrow (I_1(x) \Rightarrow O_1(x, s_1(x)), \ldots, I_n(x) \Rightarrow O_1(x, s_n(x))) \wedge$$
$$f(x) = F[s_1(x), \ldots s_n(x), x] \; |\!]$$
$$\Longrightarrow (I(x) \Rightarrow O(x, f(x))).$$

When this rule is applied in Isabelle the subgoals generated should correspond to the applicability conditions of the rule, the subproblems generated and the subgoal representing the composition of the subproblem solutions. The generation of these subgoals will be demonstrated in the next section. An instance of this generalised synthesis rule is the case introduction rule which has the following format.

$$[\!| \exists b.\ (b(x) \vee \text{not } b(x)) \wedge$$
$$I(x) \Rightarrow (b(x) \Rightarrow t(x) \wedge \text{not } b(x) \Rightarrow e(x)) \wedge$$
$$f(x) = \text{if } b(x) \text{ then } O(x, t(x)) \text{ else } O(x, e(x))) \; |\!]$$
$$\Longrightarrow I(x) \Rightarrow O(x, f(x)).$$

The applicability condition $b(x) \vee \text{not } b(x)$ is trivially true. The subproblems generated are the definitions of t(x), e(x) and f(x) . The function t(x) represents the solution where b(x) is true. The function e(x) represents the solution where b(x) is not true and f(x) represents the combination of all subproblem solutions to achieve the overall solution. Note that the application of this rule will require the instantiation of the guard b. This must be explicitly chosen by the user as it is a design decision determining the conditions under which a case distinction will be made. This is the reason $\exists$ b. is present in the above rule and this will be discussed in the section which follows.

Another example of a synthesis rule is the embedding rule which permits generalisation of a problem. If the original problem is not easily solved then it may be easier to introduce a new function to solve a more general version of the problem. The original problem is solved by instantiating the new functions parameters so that its definition is equivalent to the original specifications. An example is when a constant is replaced by a variable. This extends the problem to the range of permitted values for the variable. The original problem is solved by solving the new function with the instantiation of the variable to the original constants value. Therefore the format of the embedding rule is

$$[\!| \text{Applicability Conditions} \wedge$$
$$\text{Generalised function definition} \wedge$$
$$\text{Original function call} = \text{Generalised function call instantiated to original specification} \; |\!]$$
$$\longrightarrow \text{Original function definition}$$

The application of this rule to the N Queens problem will be seen in the next section.

Application of a Synthesis Rule:

A program is constructed from a specification by the application of synthesis rules. On an Isabelle level a synthesis rule is applied using the resolution tactic br. This tactic takes the synthesis rule name as a parameter and applies it to a named subgoal. The effect of resolution is that the problem is resolved to a conjunction of subproblems and program fragments which correspond to the instantiated version of the left hand side of the applied rule.

This conjuction of terms may require user interaction to instantiate variables whose values are unknown. A typical example of where a user may have to instantiate is where a design decision has been made to introduce a case distinction. User instantiations have no counterpart in theorem proving. The user must instantiate explicitly where instantiation is not done automatically. The proof state is then examined and presented to the user as a series of subgoals which may be focused on for further development. In Isabelle these steps must be carried out separately. On the programming level a transformation step should result in a program development state which is ready for another transformation step. Therefore the above actions are combined into one programming command.

The detailed Isabelle steps in the application of synthesis rules are now presented. These are followed by a packaged programming tactic which achieves the combined effect of the Isabelle steps in one programming command.

The proof state generated by the set up tactic for the N queens problem may be transformed into the proof state

1. True $\longrightarrow$ queens(n) = " $\exists$ s . ||cc(s,eseq)|| =n $\wedge$ nconf(cc(s,eseq))

as s is the same as the concatenation of the sequence s and the empty sequence eseq. This problem may be generalised by the technique of embedding as discussed in the previous section. The result of an embedding is a proof state of the following form

1. ||t|| $\leq$ n $\wedge$ nconf t $\longrightarrow$ qu (n,t) = ($\exists$s. ||cc (s, t)|| = n $\wedge$ nconf (cc (s, t)))
2. True $\longrightarrow$ queens n = qu (n,eseq)
3. qu (n,eseq) $\longrightarrow$ ($\exists$s. ||cc (s, eseq)|| = n $\wedge$ nconf (cc (s, eseq)))

where subgoal one is the new generalised problem specification. Subgoal two is the original specification defined in terms of the generalised specification with the specialisation parameter passed to it. This subgoal solves the specification if the definition of qu is in an operational form. Subgoal three is the applicability condition of the embedding rule which must be true for the embedding to be valid. This applicability condition states that the new definition instantiated to the special case implies the original problem definition. Subgoal one is the only goal which remains for development.

To apply this technique the embedding rule must be formalised within Isabelle. The application of the rule requires user interaction to supply the new function name qu , the modifications to generate the new function definition (subgoal one) and the parameters which specialise the general problem to the original one. The generalised function can be obtained from the original specification by substitution of variables for constants. The original specification is solved by obtaining an operational solution for the generalised specification and then instantiating the variables to the original constants values.

These interactions, replacement of constants by variables and generation of a proof state as above require the user to be familiar with Isabelle tactics and their application. As the user is concerned with program construction rather than theorem proving a design tactic must be introduced to conceal the Isabelle actions. The individual steps are packaged so that the user is only concerned with the concepts related to transformational programming. Hence when the user applies an embedding rule all intermediate states are hidden and the proof state generated is as above. The only interaction the user must have in the application of this rule is to supply the generalised functions name and indicate what constants to replace by variables. All other activities can be automated. The Isabelle details and the creation of a design tactic are discussed in more detail in the application of the next transformation rule in the development- the case introduction rule.

The case introduction rule is applied to subgoal one of the proof state generated as a result of the above embedding

1. ||t|| $\leq$ n $\wedge$ nconf t $\longrightarrow$ qu (n,t) = $\exists$s. ||cc (s, t)|| = n $\wedge$ nconf (cc (s, t))
2. True $\longrightarrow$ queens n = qu (n,eseq)
3. qu (n,eseq) $\longrightarrow$ ($\exists$s. ||cc (s, eseq)|| = n $\wedge$ nconf (cc (s, eseq)))

using

```
br case_intro 1;
```

where case_intro is the name of the Isabelle case introduction rule. Subgoal one matches with the left hand side of the rule and is replaced by the instantiated right hand side of the rule. This causes subgoal one - the definition of qu (n,t) be rewritten as

1. ∃ b. (b ∨ not b) ∧
   (b ⇒ ∃ ta. (nconf(t) ∧ ||t|| ≤ n) → ta t n = ∃ s . ||cc(t,s)|| =n ∧ nconf(cc(t,s)) ∧
   (not b ⇒ ∃ e. (nconf(t) ∧ ||t|| ≤ n)→ e t n = ∃ s . ||cc(t,s)|| =n ∧ nconf(cc(t,s)) ∧
   ∃ ta e. f = if b then ta else e

Note that this goal is in the form

∃ Guard. Conditions ∧
Guard ⇒ Subgoal1 ∧
Not Guard ⇒ Subgoal2 ∧
f = if Guard then Solution1 else Solution2

This resolution step generates an intermediate state in the development process. The case introduction rule has been applied but the goal is not in a state where the user can easily focus on part of it to continue the development.

The variables b , ta and e are uninstantiated. The guard b must be instantiated by the programmer as the choice of guard is a design decision in program construction. It determines the conditions under which the case distinction is carried out. One possibility for the instantiation of the guard in this example is ||t ||= n . To instantiate b we must replace it with ||t ||= n in subgoal 1 above. This is achieved in Isabelle by instantiating the rule exI

P x ⟹ ∃ x. P x

The Isabelle tactic res_inst_tac is used to instantiate the bound variable x in the above rule. This leads to

P (||t|| = n) ⟹ ∃ x. P x

which may be resolved with subgoal one to achieve the proof state

1. (||t|| = n ∨ ||t|| ≠ n) ∧
   (||t|| = n ⇒ (∃ ta. nconf(t) ∧ ||t|| ≤ n)→ ta t n = (∃ s . (||cc(t,s)|| =n ∧ nconf(cc(t,s))) ∧
   (||t|| ≠ n ⇒ (∃ e. (nconf(t) ∧ ||t|| ≤ n)→ e t n = (∃ s . (||cc(t,s)|| =n ∧ nconf(cc(t,s)))) ∧
   (∃ ta e. f = (if ||t||= n then ta else e))

This two steps may be combined by the Isabelle tactic

```
by (res_inst_tac [("x", "||t||=n")] exI 1);
```

The functions ta and e represent the solutions to the subproblems generated by application of the synthesis rule. These are developed by the application of transformation rules using the additional information supplied by the guard. To develop these functions it must be possible to focus in on each definition. Therefore, it should be possible to split the proof state generated by the application of a synthesis rule into different subgoals. This generates a proof state which is ready for further development.

The rule conjI is a rule in Isabelle which states that

[| P; Q |] ⟹ P ∧ Q.

When this rule is resolved with a conjunction of terms it splits of one conjunct and numbers it as a separate subgoal. This enables focusing on parts of the term for further development.

The application of the case introduction rule, the user instantiation of the guard and the subgoal organisation is done by the following list of Isabelle commands.

```
br case_intro 1; ( Introduce case distinction )
by (res_inst_tac [("x", "||t||=n")] exI 1); ( Instantiate guard of if statement )
br conjI 1;
br conjI 2;
br conjI 3; ( Split into subgoals corresponding to conjuncts )
```

The output from Isabelle as a result of the above resolution steps is

1. $||t|| = n \vee ||t|| \neq n$
2. $||t|| = n \longrightarrow (\exists\ ta.\ (nconf(t) \wedge ||t|| \leq n) \rightarrow ta\ t = \exists\ s\ .\ ||cc(t,s)|| = n \wedge nconf(cc(t,s))$
3. $||t|| \neq n \longrightarrow (\exists\ e.\ (nconf(t) \wedge ||t|| \leq n) \rightarrow e\ t = \exists\ s\ .\ ||cc(t,s)|| = n \wedge nconf(cc(t,s))$
4. $\exists\ ta\ e.\ f = (if\ ||t|| = n\ then\ ta\ else\ e)$

It is now possible to apply synthesis rules, simple rewrite rules and Isabelle tactics to the individual subgoals. The case introduction rule generates the applicability condition

Guard $\vee$ Not Guard

which is trivially true. This subgoal can be discharged by the application of the Isabelle tactic fast_tac which applies the applicable rules from a set of rules given to it as a parameter i.e. by (fast_tac HOL_cs 1) applies the rules from the classification set HOL_cs to subgoal 1 simplifying it to true. When a goal is simplified to true it disappears from the proof state.

In the development as presented, the user is concerned with the Isabelle layer of activities. It is clear from the example that one cannot demand such detailed knowledge of the theorem proving paradigm from a programmer. Instead, a design tactic must be introduced to hide the Isabelle actions from the programmer. The individual steps are packaged so that the user is only concerned with the transformation step e.g. the application of the case introduction rule.

In our example the only information required from the user is the term representing the guard and the subproblem which the rule is applied to. All other actions can be done automatically. Therefore the case introduction rule may be packaged in a design tactic as follows

```
fun CASE(g,n) = (
br case_intro n;
by (res_inst_tac [("x",g)] exI n);
br conjI n;
br conjI (n + 1);
br conjI (n + 2);
by (fast_tac HOL_cs 1)
);
```

This tactic is invoked to apply the case introduction rule as detailed above, by writing

```
CASE (||t|| = n, 1);
```

where $||t|| = n$ is the user supplied instantiation of the guard and 1 is the number of the subgoal to which we want to apply the rule.

A design tactic like this must be written for all the synthesis rules. The design tactics have a synthesis rule as their logical core which are embedded in basic Isabelle tactics. These Isabelle tactics modify the intermediate states into an acceptable program development state for presentation to the programmer. Now the programmers only concern is

the decisions of which rules to apply, which subgoal to apply it to and what instantiation to give parameters associated with a rules application.

As previously stated the construction of DEBATE is continuation of a project in which a Prolog based tool for program construction was developed. This Prolog based tool assisted the user in choosing which rules to apply to a problem specification by presenting the applicable rules to the user in menu format. A dialogue approach was used to indicate which rule to apply and to instantiate unknown variables in that rule. However, this presentation of applicable rules was too general and often resulted in numerous rules being suggested. This approach can also be taken in Isabelle using the findI tactic which lists all the applicable rules for a particular subgoal. It is more desirable, however, to have a precise suggestion as to which rule to apply. This can be achieved by studying the problem specification and the type of solution which is required by the user. A more advanced tool would supply a knowledge based approach in which tactical knowledge is encoded. This would greatly assist the automation of the process of program construction by transformation. However, this is a difficult problem and is outside the scope of the current paper.

3.3 Simplification

Simplification is carried out in program construction by using the simple rewrite rules discussed earlier. These rules permit the rewriting of specifications without the introduction of subproblems. Isabelle provides a suite of simplification tactics which permit both conditional and unconditional rewriting. Usually in theorem proving we aim to simplify all subgoals to true so that the original goal is achieved. In program construction we do not wish this to happen as the goals remaining at the end of the development represent the solution to the original problem. Therefore the user must be careful when they use the simplifier so that the subgoal which would have lead to a solution is not discarded.

A base simplification set is established for use during the program construction. Applying the simplifier attempts the application of the rules from this set to the goal being developed. Rules may be added to and removed from this set during a construction. The order in which simplification rules are applied is important as it determines the form of the final solution. The order of application can be controlled by the order in which rules are added to the simplification set. Isabelle provides many of the simplification rules required during program construction. Other simplification rules are obtained from the problem specification and from rules which the user proves themselves. Tactics have been set up to allow application of all the rules in the simplification set as well as rules added from the specification axioms and rules the user has proved themselves.

3.4 Closing Developments

In certain circumstances the solution to a subproblem may be obvious. Rather than working forward with the development through the application of transformation rules to a specification, it is often easier to verify that a proposed solution is correct. A solution may be suggested and its correctness is proved with respect to the specifications developed so far. Note that this process is a backwards proof rather than the forward developments we have been dealing with so far. If the suggested solution is proved correct it is accepted as that subproblems solution.

Suppose the problem presented is of the form

$$I(x) \Rightarrow O(x, f(x)).$$

The development branch can be closed by proving that the proposed solution is adequate. Proposing a solution means that the user suggests a term of the form $f(x) = T(x)$ where $T(x)$ is an explicit description of the solution under the condition $I(x)$. This hypothesis must be checked by proving the lemma

$$I(x) \Rightarrow O(x, T(x)).$$

When this is proven correct a new definition of f is available i.e.

$$I(x) \Rightarrow (f(x) = T(x)).$$

Consequently, this may be accepted as a valid law solution to the original specification. Closing all development branches result in the generation of a series of operational solutions. These solutions may be combined into a functional program by unfolding and simplification. These programs may be optimized by use of efficiency transformations.

In order to close a development in Isabelle we must suspend the main program construction and commence the proof of the lemma stating that the proposed solution satisfies the development. This is achieved in Isabelle by stating

```
val saved = save_proof();
```

The lemma is checked and asserted into the theory if it is proved correct. Otherwise the development is continued. When the development is closed the main proof is restarted from where it was suspended. This is achieved by writing

```
restore_proof(saved); choplev(n);
```

which restores the main development to level n. Now the additional knowledge proved in the lemma is available.

As the user should not be concerned with Isabelle details a transformation tactic is used to maintain a programming view on the development. The tactic is called the Close tactic and is written as

```
fun Close (I(x) ⇒ O(x, f(x)), T(x),n ) = (
val saved = save_proof();
if Prove(I(x) ⇒ O(x, T(x))) then Assert (f(x) = T(x))
restore_proof(saved);
choplev(n);
);
```

where $I(x) \Rightarrow O(x, f(x))$ is the problem, $T(x)$ is the proposed solution and n is the place where the main development must resume. The function Prove above causes its parameter to be set up as a subgoal for the proof to be carried out. The function Assert adds the lemma to the theory if it is proved correct.

The effect of this is the systematic extension of the original specification by propositions. These propositions express the original implicit functions in an operational manner. Since every branch of the development is closed in this way the result is an operational solution to the overall problem. Branches are closed as above where solutions are proposed and verified or may be closed automatically by the developments been pushed sufficiently far.

4 Conclusion

In this paper the similarities between transformational programming and theorem proving have been discussed and hence a basis for the use of the theorem prover Isabelle in the construction of the transformation tool DEBATE has been established.

Our goal is a program construction tool DEBATE which will be used in conjunction with the specification tool ASPECT. The interaction with DEBATE should be in line with the transformational programming paradigm. Direct interaction with the theorem proving environment is not feasible as that is the interaction within a theorem proving paradigm.

Therefore, we presented aspects of transformation tactics which concerns the programmer with the application of programming techniques. The programming tactics correspond to compound transformational steps. The parameters which must be passed to these tactics represent explicit design decisions which the programmer must make. We have explored the use of Isabelle as a transformational programming tool through doing some case studies. The knowledge obtained from this practical experience is being used to generate the transformational tactics required. These tactics are currently being implemented in University College Dublin to achieve the goals outlined above.

References

[1] Bauer, F.L , et al. : The CIP Language Group. *The Munich Project CIP. Vol I : The Wide Spectrum Language CIP-L.* LNCS 183, Springer Verlag. (1985)

[2] Partsch, H.A. : *Specification and Transformation of Programs* Springer Verlag , New York, (1990)

[3] Feather, M.S. : A System for assisting program transformation. *ACM Trans. Program. Lang. Syst 4*, 1, (1982), 1-20

[4] Hoffman, B., Krieg-Bruckner. B., Eds. : *Program Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System.* LNCS 680, Springer Verlag. (1993)

[5] Wolff, B. et al. : Proving Transformations in Isabelle, Universitat Bremen, (1994)

[6] Smith, D.R. : KIDS - a knowledge-based Software Development System. In *Automating Software Design*. M.M. Lowry and R.D. McCartney, Eds., Menlo Park, Ca., AAAT Press/MIT Press, (1991), 483-514

[7] Smith, D.R. : KIDS - A Semi-Automatic Program Development System. In *IEEE Transactions on Software Engineering Special Issue on Formal Methods.* Kestrel Institute, Ca., (1990)

[8] Smith, D.R. : Towards a Classification Approach to Degign, Submitted to AMAST '96, Kestrel Institute, Ca., (1996)

[9] Toomey, C. : ASPECT: An Algebraic Specification Construction Tool. TR-95-3, Department of Computer Science, UCD, (1995)

[10] Geiselbrechtinger, F. : Structuring Specifications. TR-95-2, Department of Computer Science, UCD, (1995)

[11] Monahan, R. : Transformational Programming. Department of Computer Science, UCD, (1995)

[12] Manna, Z., Waldinger R. : Fundamentals of deductive Program Synthesis. In *Logic, Algebra and Computation,* F.L. Bauer, Ed., Springer Verlag, (1991), 41-107.

[13] Monahan, R. : Deduction Based Transformational Programming, Department of Computer Science, UCD, (in preparation)

[14] Monahan, R. , F Geiselbrechtinger : Theorem Proving and Transformational Programming, submitted to FME '97 Department of Computer Science, UCD. (1997)

[15] Paulson, L. : Introduction to Isabelle, University of Cambridge, (1995)

[16] Paulson, L. : Isabelle object logics, University of Cambridge, (1995)

[17] Paulson, L. : The Isabelle Reference Manual, University of Cambridge, (1995)

[18] Kalvala, S. : A Gentle Introduction to Isabelle, University of Cambridge, (1995)

[19] Geiselbrechtinger, F. : SPL: A Notation for Structured Specifications TR-94-7. Department of Computer Science, UCD, (1994)